<https://leetcode.com/problems/letter-tile-possibilities/>

You have n  tiles, where each tile has one letter tiles[i] printed on it.

Return *the number of possible non-empty sequences of letters* you can make using the letters printed on those tiles.

**Example 1:**

Input: tiles = "AAB"

Output: 8

Explanation: The possible sequences are "A", "B", "AA", "AB", "BA", "AAB", "ABA", "BAA".

**Example 2:**

Input: tiles = "AAABBC"

Output: 188

**Example 3:**

Input: tiles = "V"

Output: 1

**Constraints:**

* 1 <= tiles.length <= 7
* tiles consists of uppercase English letters.

**Attempt 1: 2023-08-02**

**Wrong Solution: No duplicate handling if only refer to L46.Permutations, have to introduce logic from L47.Permutations II to handle duplicates**

class Solution {

public int numTilePossibilities(String tiles) {

List<List<Character>> result = new ArrayList<>();

char[] chars = tiles.toCharArray();

boolean[] visited = new boolean[chars.length];

for(int i = 1; i <= chars.length; i++) {

helper(chars, result, new ArrayList<Character>(), i, visited);

}

return result.size();

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private void helper(char[] chars, List<List<Character>> result, List<Character> tmp, int k, boolean[] visited) {

if(tmp.size() == k) {

result.add(new ArrayList<Character>(tmp));

return;

}

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

if(visited[i]) {

continue;

}

visited[i] = true;

tmp.add(chars[i]);

helper(chars, result, tmp, k, visited);

visited[i] = false;

tmp.remove(tmp.size() - 1);

}

}

}

**Solution 1: Backtracking with actual storage of all combinations of permutations (10min)**

class Solution {

public int numTilePossibilities(String tiles) {

List<List<Character>> result = new ArrayList<>();

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

for(int i = 1; i <= chars.length; i++) {

helper(chars, result, new ArrayList<Character>(), i, visited);

}

return result.size();

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private void helper(char[] chars, List<List<Character>> result, List<Character> tmp, int k, boolean[] visited) {

if(tmp.size() == k) {

result.add(new ArrayList<Character>(tmp));

return;

}

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

visited[i] = true;

tmp.add(chars[i]);

helper(chars, result, tmp, k, visited);

visited[i] = false;

tmp.remove(tmp.size() - 1);

}

}

}

**Refer to**

<https://leetcode.com/problems/permutations-ii/discuss/18594/Really-easy-Java-solution-much-easier-than-the-solutions-with-very-high-vote/324818>

The difficulty is to handle the duplicates.

With inputs as [1a, 1b, 2a],

If we don't handle the duplicates, the results would be: [1a, 1b, 2a], [1b, 1a, 2a]...,

so we must make sure 1a goes before 1b to avoid duplicates

By using nums[i-1]==nums[i] && !used[i-1], we can make sure that 1b cannot be chosen before 1a

<http://www.jiuzhang.com/solutions/permutations-ii/>

public class Solution {

public List<List<Integer>> permuteUnique(int[] nums) {

List<List<Integer>> result = new ArrayList<List<Integer>>();

if(nums == null || nums.length == 0) {

return result;

}

Arrays.sort(nums);

List<Integer> combination = new ArrayList<Integer>();

helper(nums, result, combination, new boolean[nums.length]);

return result;

}

private void helper(int[] nums, List<List<Integer>> result, List<Integer> combination, boolean[] used) {

if(combination.size() == nums.length) {

result.add(new ArrayList<Integer>(combination));

}

for(int i = 0; i < nums.length; i++) {

/\*

判断主要是为了去除重复元素影响。

比如，给出一个排好序的数组，[1,2,2]，那么第一个2和第二2如果在结果中互换位置，

我们也认为是同一种方案，所以我们强制要求相同的数字，原来排在前面的，在结果

当中也应该排在前面，这样就保证了唯一性。所以当前面的2还没有使用的时候，就

不应该让后面的2使用。

\*/

if(used[i] || (i > 0 && !used[i - 1] && nums[i] == nums[i - 1])) {

continue;

}

used[i] = true;

combination.add(nums[i]);

helper(nums, result, combination, used);

combination.remove(combination.size() - 1);

// Don't forget to reset 'used' boolean flag back to false

used[i] = false;

}

}

}

**Solution 2: Backtracking without actual storage of all combinations of permutations (120min)**

**And since we only care about the count of all combinations of letters for sizes 1...n, and aggregate permutations of each combination,  we can remove the actual permutations of combinations and just calculate the count, similar to L60. Permutation Sequence, change helper() method return type and construct recursion logic to aggregate total count based on all recursion levels**

**第一轮升级: 由于不再需要真实记录所有的combination的permutation，移除List<List<Character>> result，改变return的类型，从void变成int，base condition("底")的条件依然保持为if(tmp.size() == k)但是不再需要加入tmp到result中，而是直接返回1，表示找到一个满足k个字符的解，然后新设定一个本地递归本地变量count记录每一层返回当前层的combination的permutation的个数，然后继续累加更深层的递归的结果，类似L940中**ans += 1 + helper(arr, i + 1, lvl + 1)**的操作，最后在递归方法结果出口(不是base condition("底")的出口，而是所有递归完成之后的出口)直接返回count就行**

class Solution {

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

int result = 0;

for(int i = 1; i <= chars.length; i++) {

result += helper(chars, new ArrayList<Character>(), i, visited);

}

return result;

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private int helper(char[] chars, List<Character> tmp, int k, boolean[] visited) {

// Base condition

// Return 1 means find a solution -> "a permutation of combination has k chars"

if(tmp.size() == k) {

return 1;

}

// Add a new local variable for recursion method to record all levels's

// combination of permutations number

int count = 0;

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

visited[i] = true;

tmp.add(chars[i]);

// Similar to L940 way aggregate all recursion levels' count into one place

count += helper(chars, tmp, k, visited);

visited[i] = false;

tmp.remove(tmp.size() - 1);

}

// The final return is the total count

return count;

}

}

**第二轮升级: 去掉for(int i = 1; i <= chars.length; i++) {helper(...)}，不用for循环进入递归方法helper(...)而是直接进入单一helper(...)但是同样实现for循环所做到的找所有combination长度下的permutation个数的效果，实现过程中需要做一些改进，(1)因为不需要记录实际的所有combination长度下的permutation，去掉List<List<Character>> result和List<Character> tmp参数，(2)去掉配合for循环使用的代表需要从给定字符串中选择多少个字符的参数k，(3)要找到所有combination长度(=k)下的permutation就必须去掉递归方法中到"底"时需要满足的条件，比如if(tmp.size() == k) {result += ...}，因为所有combination长度(=k)下都满足加入总和结果(=result)的条件，也就不再需要if(tmp.size() == k)条件限制{result += ...}，每一层递归的计数直接加入总和结果(=result)就可以了，参见L90/P11.2.Subsets II，L940.Distinct Subsequences II的处理方法**

**Style 1: Recursion method local variable 'count' (和前文第一轮升级采用同样的方式)**

class Solution {

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

return helper(chars, visited);

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private int helper(char[] chars, boolean[] visited) {

// No base condition as all combinations of permutations need to record

int count = 0;

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

// 把count += 1 + ...放在具体的递归for循环中能天然避免empty subsequence

// e.g {"AB"} -> result = {{"A"},{"B"},{"AB"},{"BA"}}

visited[i] = true;

count += 1 + helper(chars, visited);

visited[i] = false;

}

return count;

}

}

**Refer to**

<https://leetcode.com/problems/letter-tile-possibilities/solutions/308333/simple-java-solution-of-backtracking-no-need-for-extra-space/comments/438975/>

class Solution {

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

return dfs(chars,visited);

}

private int dfs(char[] chars, boolean[] visited){

int count = 0;

for(int i = 0; i < chars.length; i++){

if(visited[i]) continue;

if(i - 1 >= 0 && chars[i] == chars[i - 1] && !visited[i - 1]) continue;

visited[i] = true;

count++;

count+=dfs(chars, visited);

visited[i] = false;

}

return count;

}

}

**Style 2: Global variable 'count' (count所在位置会有两种不同的写法，一种需要去掉空字符，count++在原base condition所在的位置，另一种天然不包含空字符，count++在递归方法内部的for循环体重)**

**需要去掉空字符的写法**

// 'count++' directly at base condition position, will cause additional empty subsequence

// added into total count, remove by -1 in the end

class Solution {

int count = 0;

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

helper(chars, visited);

// Remove the empty subsequence

return count - 1;

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private void helper(char[] chars, boolean[] visited) {

// No base condition as all combinations of permutations need to record

// But 'count++' out of for loop recursion logic will cause adding empty

// subsequence into total count, need -1 to remove in final result

count++;

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

visited[i] = true;

helper(chars, visited);

visited[i] = false;

}

}

}

我们可以用再次加上result, tmp这两个本来已经移除的参数，用于观察count++ 放置于递归方法

base condition位置时empty subsequence的由来

======================================================================================

class Solution {

int count = 0;

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

// 再次加上result, tmp这两个本来已经移除的参数，用于观察count++

// 放置于递归方法base condition位置时empty subsequence的由来

List<List<Character>> result = new ArrayList<>();

helper(chars, result, new ArrayList<>(), visited);

// Remove the empty subsequence

return count - 1;

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private void helper(char[] chars, List<List<Character>> result, List<Character> tmp, boolean[] visited) {

// No base condition as all combinations of permutations need to record

count++;

// 基于count++位于原base condition的位置，result也直接加入本次递归中的tmp，

// 不附加任何条件，这样我们就会观测到empty subsequence每次都会在第一次递归

// 开始时首先加入result中

// e.g {"AB"} -> result = {{},{"A"},{"B"},{"AB"},{"BA"}}

result.add(new ArrayList<>(tmp));

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

visited[i] = true;

tmp.add(chars[i]);

helper(chars, result, tmp, visited);

visited[i] = false;

tmp.remove(tmp.size() - 1);

}

}

}

**Refer to**

<https://leetcode.com/problems/letter-tile-possibilities/solutions/898229/java-backtracking-solution-with-detailed-explanation/>

int count = 0;

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

Arrays.sort(chars);

backtrack(chars, new boolean[chars.length]);

// remove the empty one

return count - 1;

}

private void backtrack(char[] chars, boolean[] used) {

count = count + 1;

for (int i = 0; i < chars.length; i++) {

// if current character was used before

// OR

// previous character is equals to current character

// and previoud character was not used before

// e.g [a,a,b]

// if use first 'a' or second 'a' as the first char of the sequence

// which will result repeated permutations

if (used[i] || i > 0 && chars[i] == chars[i - 1] && !used[i - 1])

continue;

used[i] = true;

backtrack(chars, used);

used[i] = false;

}

}

**天然不包含空字符的写法**

// 'count++' in recursion method for loop body, will avoid additional empty subsequence

// added into total count, no need -1 in the end

class Solution {

int count = 0;

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

helper(chars, visited);

// 由于count++放在具体的递归for循环中，天然避免empty subsequence，

// 不再需要考虑多余的empty subsequence

return count;

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private void helper(char[] chars, boolean[] visited) {

// No base condition as all combinations of permutations need to record

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

// 把count++放在具体的递归for循环中能天然避免empty subsequence

// e.g {"AB"} -> result = {{"A"},{"B"},{"AB"},{"BA"}}

count++;

visited[i] = true;

helper(chars, visited);

visited[i] = false;

}

}

}

我们可以用再次加上result, tmp这两个本来已经移除的参数，用于观察count++ 放置于递归for循环内部

位置时天然避免empty subsequence

======================================================================================

class Solution {

int count = 0;

public int numTilePossibilities(String tiles) {

char[] chars = tiles.toCharArray();

// Add sorting like L47 to handle duplicates

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

// 再次加上result, tmp这两个本来已经移除的参数，用于观察count++

// 放置于递归方法base condition位置时empty subsequence的由来

List<List<Character>> result = new ArrayList<>();

helper(chars, result, new ArrayList<>(), visited);

// 由于count++放在具体的递归for循环中，天然避免empty subsequence，

// 不再需要考虑多余的empty subsequence

return count;

}

// Find all unique permutations based on k chars

// And if not handling duplicates well then in wrong result

// e.g "AAB"

// Expect for k = 3 chars as {"AAB", "ABA", "BAA"}

// and only use L46.Backtracking style 2 which have no duplicates handling will result in

// wrong set as {"AAB", "ABA", "AAB", "ABA", "BAA", "BAA"}

// Need to introduce L47 duplicate handling logic

private void helper(char[] chars, List<List<Character>> result, List<Character> tmp, boolean[] visited) {

// No base condition as all combinations of permutations need to record

result.add(new ArrayList<>(tmp));

// Note: The element select range still [0, chars.length - 1]

// which means still select k elements among all chars, not

// only between [0, k - 1] chars

for(int i = 0; i < chars.length; i++) {

// Add duplicate handling logic like L47

if(visited[i] || i > 0 && !visited[i - 1] && chars[i - 1] == chars[i]) {

continue;

}

// 把count++放在具体的递归for循环中能天然避免empty subsequence

// e.g {"AB"} -> result = {{"A"},{"B"},{"AB"},{"BA"}}

count++;

visited[i] = true;

tmp.add(chars[i]);

helper(chars, result, tmp, visited);

visited[i] = false;

tmp.remove(tmp.size() - 1);

}

}

}

**Refer to**

<https://leetcode.com/problems/letter-tile-possibilities/solutions/308333/simple-java-solution-of-backtracking-no-need-for-extra-space/>

class Solution {

int count;

public int numTilePossibilities(String tiles) {

count = 0;

char[] chars = tiles.toCharArray();

Arrays.sort(chars);

boolean[] visited = new boolean[chars.length];

dfs(chars, 0, visited);

return count;

}

private void dfs(char[] chars, int length, boolean[] visited){

if(length == chars.length) return;

for(int i = 0; i < chars.length; i++){

if(visited[i]) continue;

if(i - 1 >= 0 && chars[i] == chars[i - 1] && !visited[i - 1]) continue;

count ++;

visited[i] = true;

dfs(chars, length + 1, visited);

visited[i] = false;

}

}

}

**Refer to**

<https://leetcode.com/problems/letter-tile-possibilities/solutions/308323/java-1ms-simple-solution-which-combines-permutation-and-subset/>

This problem is the combination of [90. Subsets II](https://leetcode.com/problems/subsets-ii/) and [47. Permutations II](https://leetcode.com/problems/permutations-ii/). What we need to calculate is the number of distinct sub-permutation of input array. The key is removing duplicate elements and counting the number of distinct sub-permutation:

* Duplicate removal is the same as [47. Permutations II](https://leetcode.com/problems/permutations-ii/discuss/278301/java-1ms-solution-which-beats-100-with-explanation-of-how-to-remove-duplicate) where we need to sort the array first and judge whether current element is equal to the previous one.
* Number counting is the same as [90. Subsets II](https://leetcode.com/problems/subsets-ii/discuss/308274/java-1ms-simple-solution) where we count the number at the beginning of each round of traversal.